5

Operators and Casts

WHAT'S IN THIS CHAPTER?

- Operators in C#
- Implicit and explicit conversions
- Overloading standard operators for custom types
- Comparing objects for equality
- Implementing custom indexers
- User-defined conversions

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/OperatorsAndCasts`.

The code for this chapter is divided into the following major examples:

- OperatorsSample
- BinaryCalculations
- OperatorOverloadingSample
- EqualitySample
- CustomIndexerSample
- UserDefinedConversion

All the projects have nullable reference types enabled.

The preceding chapters have covered most of what you need to start writing useful programs using C#. This chapter continues the discussion with essential language elements and illustrates some powerful aspects of C# that enable you to extend its capabilities. This chapter also covers information about using operators and extending custom types with operator overloading and custom conversion.

OPERATORS

C# supports the operators and expressions listed in the following table. In the table, the operators start with the highest precedence and go down to the lowest.

CATEGORY	OPERATOR
Primary	`x.y x?.y f(x) a[x] x++ x-- x! x->y new typeof default checked unchecked delegate nameof sizeof delegate stackalloc`
Unary	`+x -x !x ~x ++x --x ^x (T)x await &x *x true false`
Range	`x..y`
Multiplicative	`x*y x/y x%y`
Additive	`x+y x-y`
Shift	`x<<y x>>y`
Relational	`x<y x>y x<=y x>=y`
Type testing	`is as`
Equality	`x==y x!=y`
Logical	`x&y x^y x|y`
Conditional logical	`x&&y x||y`
Null coalescing	`x??y`
Conditional operator	`c?t:f`
Assignment	`x=y x+=y x-=y x*=y x/=y x%=y x&=y x|=y x^=y x<<=y x>>=y x??=y`
Lambda expression	`=>`

> **NOTE** *Four specific operators (`sizeof`, `*`, `->`, and `&`) are available only in unsafe code (code that bypasses C#'s type-safety checking), which is discussed in Chapter 13, "Managed and Unmanaged Memory."*
>
> *Using the new range and hat operators with strings is covered in Chapter 2, "Core C#." Using these operators with arrays is covered in Chapter 6, "Arrays," where you also can read how to support custom collections with these operators.*

Compound Assignment Operators

Compound assignment operators are a shortcut to using the assignment operator with another operator. Instead of writing `x = x + 2`, you can use the compound assignment `x += 2`. Incrementing by `1` is required even more often, so there's another shortcut, `x++`:

```
int x = 1;
int x += 2; // shortcut for int x = x + 2;
x++; // shortcut for x = x + 1;
```

Shortcuts can be used with all the other compound assignment operators. A new compound assignment operator has been available since C# 8: the null-coalescing compound assignment operator. This operator is discussed later in this chapter.

You may be wondering why there are two examples for the `++` increment operator. Placing the operator *before* the expression is known as a *prefix*; placing the operator *after* the expression is known as a *postfix*. Note that there is a difference in the way they behave.

The increment and decrement operators can act both as entire expressions and within expressions. When used by themselves, the effect of both the prefix and postfix versions is identical and corresponds to the statement `x = x + 1`. When used within larger expressions, the prefix operator increments the value of `x` *before* the expression is evaluated; in other words, `x` is incremented, and the new value is used as the result of the expression. Conversely, the postfix operator increments the value of `x` *after* the expression is evaluated. The result of the expression returns the original value of `x`. The following example uses the increment operator (`++`) as an example to demonstrate the difference between the prefix and postfix behavior (code file `OperatorsSample/Program.cs`):

```
void PrefixAndPostfix()
{
  int x = 5;
  if (++x == 6) // true - x is incremented to 6 before the evaluation
  {
    Console.WriteLine("This will execute");
  }
  if (x++ == 6) // true - x is incremented to 7 after the evaluation
  {
    Console.WriteLine("The value of x is: {x}"); // x has the value 7
  }
}
```

The following sections look at some of the commonly used and new operators that you will frequently use within your C# code.

The Conditional-Expression Operator (?:)

The conditional-expression operator (`?:`), also known as the *ternary operator*, is a shorthand form of the `if...else` construction. It gets its name from the fact that it involves three operands. It allows you to evaluate a condition, returning one value if that condition is true or another value if it is false. The syntax is as follows:

```
condition ? true_value: false_value
```

Here, `condition` is the Boolean expression to be evaluated, `true_value` is the value that is returned if `condition` is true, and `false_value` is the value that is returned otherwise.

When used sparingly, the conditional-expression operator can add a dash of terseness to your programs. It is especially handy for providing one of a couple of arguments to a function that is being invoked. You can use it

to quickly convert a Boolean value to a string value of `true` or `false`. It is also handy for displaying the correct singular or plural form of a word (code file `OperatorsSample/Program.cs`):

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man": "men");
Console.WriteLine(s);
```

This code displays `1 man` if `x` is equal to one but displays the correct plural form for any other number. Note, however, that if your output needs to be localized to different languages, you have to write more sophisticated routines to take into account the different grammatical rules of different languages. Read Chapter 22, "Localization," for globalizing and localizing .NET applications.

The checked and unchecked Operators

Consider the following code:

```
byte b = byte.MaxValue;
b++;
Console.WriteLine(b);
```

The `byte` data type can hold values only in the range 0 to 255. Assigning `byte.MaxValue` to a byte results in 255. With 255, all bits of the 8 available bits in the byte are set: 11111111. Incrementing this value by one causes an overflow and results in 0.

To get exceptions in such cases, C# provides the `checked` and `unchecked` operators. If you mark a block of code as `checked`, the CLR enforces overflow checking, throwing an `OverflowException` if an overflow occurs. The following changes the preceding code to include the `checked` operator (code file `OperatorsSample/Program.cs`):

```
byte b = 255;
checked
{
  b++;
}
Console.WriteLine(b);
```

Instead of writing a `checked` block, you also can use the `checked` keyword in an expression:

```
b = checked(b + 3);
```

When you try to run this code, the `OverflowException` is thrown.

You can enforce overflow checking for all unmarked code by adding the `CheckForOverflowUnderflow` setting in the `csproj` file:

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0</TargetFramework>
  <Nullable>enable</Nullable>
  <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
</PropertyGroup>
```

With a project setting to be configured for overflow checking, you can mark code that should not be checked using the `unchecked` operator.

> **NOTE** *By default, overflow and underflow are not checked because enforcing checks has a performance impact. When you use checked as the default setting with your project, the result of every arithmetic operation needs to be verified regardless of whether the value is out of bounds.* `i++` *is an arithmetic operation that's used a lot with* `for` *loops. To avoid having this performance impact, it's better to keep the default setting (Check for Arithmetic Overflow/Underflow) unchecked and use the* `checked` *operator where needed.*

The is and as Operators

You can use the `is` and `as` operators to determine whether an object is compatible with a specific type. This is useful with class hierarchies.

Let's assume a simple class hierarchy. The class `DerivedClass` derives from the class `BaseClass`. You can assign a variable of type `DerivedClass` to a variable of type `BaseClass`; all the members of the `BaseClass` are available with the `DerivedClass`. In the following example, an implicit conversion is taking place:

```
BaseClass = new();
DerivedClass = new();
baseClass = derivedClass;
```

If you have a parameter of the `BaseClass` and want to assign it to a variable of the `DerivedClass`, implicit conversion is not possible. To the `SomeAction` method, an instance of the `BaseClass` or any type that derives from this class can be passed. This will not necessarily succeed. Here, you can use the `as` operator. The `as` operator either returns a `DerivedClass` instance (if the variable is of this type) or returns `null`:

```
public void SomeAction(BaseClass baseClass)
{
  DerivedClass? derivedClass = baseClass as DerivedClass;
  if (derivedClass != null)
  {
    // use the derivedClass variable
  }
}
```

Instead of using the `as` operator, you can use the `is` operator. The `is` operator returns `true` if the conversion succeeds; otherwise, it returns `false`. With the `is` operator, a variable can be specified that is assigned if the `is` operator returns true:

```
public void SomeAction(BaseClass baseClass)
{
  if (baseClass is DerivedClass derivedClass)
  {
    // use the derivedClass variable
  }
}
```

> **NOTE** *Chapter 2 covers pattern matching with the* `is` *operator using const, type, and relational patterns.*

The sizeof Operator

You can determine the size (in bytes) required on the stack by a value type using the `sizeof` operator (code file `OperatorsSample/Program.cs`):

```
Console.WriteLine(sizeof(int));
```

This displays the number `4` because an `int` is 4 bytes long.

You can also use the `sizeof` operator with structs if the struct contains only value types—for example, the `Point` class as shown here (code file `OperatorsSample/Point.cs`):

```
public readonly struct Point
{
  public Point(int x, int y) => (X, Y) = (x, y);

  public int X { get; }
  public int Y { get; }
}
```

> **NOTE** *You cannot use* `sizeof` *with classes.*

When you use `sizeof` with custom types, you need to write the code within an unsafe code block (code file `OperatorsSample/Program.cs`):

```
unsafe
{
  Console.WriteLine(sizeof(Point));
}
```

> **NOTE** *By default, unsafe code is not allowed. You need to specify the* `AllowUnsafeBlocks` *in the* `csproj` *project file. Chapter 13 looks at unsafe code in more detail.*

The typeof Operator

The `typeof` operator returns a `System.Type` object representing a specified type. For example, `typeof(string)` returns a `Type` object representing the `System.String` type. This is useful when you want to use reflection to find information about an object dynamically. For more information, see Chapter 12, "Reflection, Metadata, and Source Generators."

The nameof Expression

The `nameof` operator is of practical use when strings are needed as parameters that are already known at compile time. This operator accepts a symbol, property, or method and returns the name.

One use example is when the name of a variable is needed, as in checking a parameter for null, as shown here:

```
public void Method(object o)
{
  if (o == null) throw new ArgumentNullException(nameof(o));
}
```

Of course, it would be similar to throw the exception by passing a string instead of using the `nameof` operator. However, using `nameof` prevents you from misspelling the parameter name when you pass it to the exception's constructor. Also, when you change the name of the parameter, you can easily miss changing the string passed to the `ArgumentNullException` constructor. Refactoring features also help changing all occurrences where `nameof` is used:

```
if (o == null) throw new ArgumentNullException("o");
```

Using the `nameof` operator for the name of a variable is just one use case. You can also use it to get the name of a property—for example, for firing a change event (using the interface `INotifyPropertyChanged`) in a property `set` accessor and passing the name of a property.

```
public string FirstName
{
  get => _firstName;
  set
  {
    _firstName = value;
    OnPropertyChanged(nameof(FirstName));
  }
}
```

The `nameof` operator can also be used to get the name of a method. This also works if the method is overloaded because all overloads result in the same value: the name of the method.

```
public void Method()
{
  Log($"{nameof(Method)} called");
```

The Indexer

You use the indexer (brackets) for accessing arrays in Chapter 6. In the following code snippet, the indexer is used to access the third element of the array named `arr1` by passing the number 2:

```
int[] arr1 = {1, 2, 3, 4};
int x = arr1[2]; // x == 3
```

Similarly to accessing elements of an array, the indexer is implemented with collection classes (discussed in Chapter 8, "Collections").

The indexer doesn't require an integer within the brackets. Indexers can be defined with any type. The following code snippet creates a generic dictionary where the key is a `string` and the value an `int`. With dictionaries, the key can be used with the indexer. In the following sample, the string `first` is passed to the indexer to set this element in the dictionary, and then the same string is passed to the indexer to retrieve this element:

```
Dictionary<string, int> dict = new();
dict["first"] = 1;
int x = dict["first"];
```

> **NOTE** *Later in this chapter in the "Implementing Custom Indexers" section, you can read how to create index operators in your own classes.*

The Null-Coalescing and Null-Coalescing Assignment Operators

The null-coalescing operator (`??`) provides a shorthand mechanism to cater to the possibility of `null` values when working with nullable and reference types. The operator is placed between two operands—the first operand must be a nullable type or reference type, and the second operand must be of the same type as the first

or of a type that is implicitly convertible to the type of the first operand. The null-coalescing operator evaluates as follows:

- If the first operand is not `null`, then the overall expression has the value of the first operand.
- If the first operand is `null`, then the overall expression has the value of the second operand.

Here's an example:

```
int? a = null;
int b;
b = a ?? 10; // b has the value 10
a = 3;
b = a ?? 10; // b has the value 3
```

If the second operand cannot be implicitly converted to the type of the first operand, a compile-time error is generated.

The null-coalescing operator is not only important with nullable types but also with reference types. In the following code snippet, the property `Val` returns the value of the `_val` variable only if it is not null. In case it is null, a new instance of `MyClass` is created, assigned to the `_val` variable, and finally returned from the property. This second part of the expression within the `get` accessor only happens when the variable `_val` is null:

```
private MyClass _val;
public MyClass Val
{
  get => _val ?? (_val = new MyClass());
}
```

Using the null-coalescing assignment operator, the preceding code can now be simplified to create a new `MyClass` and assign it to `_val` if `_val` is `null`:

```
private MyClass _val;
public MyClass Val
{
  get => _val ??= new MyClass();
}
```

The Null-Conditional Operator

The *null-conditional operator*, is a feature of C# that reduces the number of code lines. A great number of code lines in production code verify null conditions. Before accessing members of a variable that is passed as a method parameter, the variable needs to be checked to determine whether it has a value of null. Otherwise, a `NullReferenceException` would be thrown. A .NET design guideline specifies that code should never throw exceptions of these types and should always check for null conditions. However, such checks could be missed easily. This code snippet verifies whether the passed parameter `p` is not null. In case it is null, the method just returns without continuing:

```
public void ShowPerson(Person? p)
{
  if (p is null) return;
  string firstName = p.FirstName;
  //...
}
```

Using the null-conditional operator to access the `FirstName` property (`p?.FirstName`), when `p` is `null`, only `null` is returned without continuing to the right side of the expression (code file `OperatorsSample/Program.cs`):

```
public void ShowPerson(Person? p)
{
  string firstName = p?.FirstName;
  //...
}
```

When a property of an `int` type is accessed using the null-conditional operator, the result cannot be directly assigned to an `int` type because the result can be `null`. One option to resolve this is to assign the result to a nullable `int`:

```
int? age = p?.Age;
```

Of course, you can also solve this issue by using the null-coalescing operator and defining another result (for example, `0`) in case the result of the left side is `null`:

```
int age1 = p?.Age ?? 0;
```

You also can combine multiple null-conditional operators. In the following example, the `Address` property of a `Person` object is accessed, and this property in turn defines a `City` property. Null checks need to be done for the `Person` object and, if it is not null, also for the result of the `Address` property:

```
Person p = GetPerson();
string city = null;
if (p != null && p.HomeAddress != null)
{
  city = p.HomeAddress.City;
}
```

When you use the null-conditional operator, the code becomes much simpler:

```
string city = p?.HomeAddress?.City;
```

You can also use the null-conditional operator with arrays. With the following code snippet, a `NullReferenceException` is thrown using the index operator to access an element of an array variable that is `null`:

```
int[] arr = null;
int x1 = arr[0];
```

Of course, traditional null checks could be done to avoid this exceptional condition. A simpler version uses `?[0]` to access the first element of the array. In case the result is `null`, the null-coalescing operator returns the value 0 for the `x1` variable:

```
int x1 = arr?[0] ?? 0;
```

USING BINARY OPERATORS

Working with binary values historically has been an important concept to understand when learning programming because the computer works with 0s and 1s. Many people who are newer to programming may have missed learning this because they start to learn programming with Blocks, Scratch, Python, and possibly JavaScript. If you are already fluent with 0s and 1s, this section might still help you as a refresher.

First, let's start with simple calculations using binary operators. The method `SimpleCalculations` first declares and initializes the variables `binary1` and `binary2` with binary values—using the binary literal and

digit separators. Using the `&` operator, the two values are combined with the binary `AND` operator and written to the variable `binaryAnd`. In the following code, the `|` operator is used to create the `binaryOr` variable, the `^` operator for the `binaryXOR` variable, and the `~` operator for the `reverse1` variable (code file `BinaryCalculations/Program.cs`):

```
void SimpleCalculations()
{
  Console.WriteLine(nameof(SimpleCalculations));
  uint binary1 = 0b1111_0000_1100_0011_1110_0001_0001_1000;
  uint binary2 = 0b0000_1111_1100_0011_0101_1010_1110_0111;
  uint binaryAnd = binary1 & binary2;
  DisplayBits("AND", binaryAnd, binary1, binary2);
  uint binaryOR = binary1 | binary2;
  DisplayBits("OR", binaryOR, binary1, binary2);
  uint binaryXOR = binary1 ^ binary2;
  DisplayBits("XOR", binaryXOR, binary1, binary2);
  uint reverse1 = ~binary1;
  DisplayBits("NOT", reverse1, binary1);
  Console.WriteLine();
}
```

To display `uint` and `int` variables in a binary form, the extension method `ToBinaryString` is created. `Convert.ToString` offers an overload with two `int` parameters, where the second `int` value is the `toBase` parameter. Using this, you can format the output string binary by passing the value 2 (for binary), 8 (for octal), 10 (for decimal), and 16 (for hexadecimal). By default, if a binary value starts with 0 values, these values are ignored and not printed. The `PadLeft` method fills up these 0 values in the string. The number of string characters needed is calculated by the `sizeof` operator and a left shift of four bits. The `sizeof` operator returns the number of bytes for the specified type, as discussed earlier in this chapter. For displaying the bits, the number of bytes needs to be multiplied by 8, which is the same as shifting three bits to the left. Another extension method is `AddSeparators`, which adds _ separators after every four digits using LINQ methods (code file `BinaryCalculations/BinaryExtensions.cs`):

```
public static class BinaryExtensions
{
  public static string ToBinaryString(this uint number) =>
    Convert.ToString(number, toBase: 2).PadLeft(sizeof(uint) << 3, '0');

  public static string ToBinaryString(this int number) =>
    Convert.ToString(number, toBase: 2).PadLeft(sizeof(int) << 3, '0');

  public static string AddSeparators(this string number) =>
    string.Join('_',
      Enumerable.Range(0, number.Length / 4)
        .Select(i => number.Substring(i * 4, 4)).ToArray());
}
```

> **NOTE** *The* `AddSeparators` *method makes use of LINQ. LINQ is discussed in detail in Chapter 9, "Language Integrated Query."*

The method `DisplayBits`, which is invoked from the previously shown `SimpleCalculations` method, makes use of the `ToBinaryString` and `AddSeparators` extension methods. Here, the operands used for the operation are displayed, as well as the result (code file `BinaryCalculations/Program.cs`):

```
void DisplayBits(string title, uint result, uint left,
  uint? right = null)
{
  Console.WriteLine(title);
  Console.WriteLine(left.ToBinaryString().AddSeparators());
  if (right.HasValue)
  {
    Console.WriteLine(right.Value.ToBinaryString().AddSeparators());
  }
  Console.WriteLine(result.ToBinaryString().AddSeparators());
  Console.WriteLine();
}
```

When you run the application, you can see the following output using the binary `&` operator. With this operator, the resulting bits are only 1 when both input values are also 1:

```
AND
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_1100_0011_0101_1010_1110_0111
0000_0000_1100_0011_0100_0000_0000_0000
```

When you apply the binary | operator, the result bit is set (1) if one of the input bits is set:

```
OR
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_1100_0011_0101_1010_1110_0111
1111_1111_1100_0011_1111_1011_1111_1111
```

With the ^ operator, the result is set if just one of the original bits is set, but not both:

```
XOR
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_1100_0011_0101_1010_1110_0111
1111_1111_0000_0000_1011_1011_1111_1111
```

And finally, with the ~ operator, the result is the negation of the original:

```
NOT
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_0011_1100_0001_1110_1110_0111
```

> **NOTE** *For working with binary values, read about using the* `BitArray` *class in Chapter 6.*

Shifting Bits

As you've already seen in the previous sample, shifting three bits to the left is a multiplication by 8. A shift by one bit is a multiplication by 2. This is a lot faster than invoking the multiply operator—in case you need to multiply by 2, 4, 8, 16, 32, and so on.

The following code snippet sets one bit in the variable `s1`, and in the `for` loop the bit always shifts by one bit (code file `BinaryCalculations/Program.cs`):

```
void ShiftingBits()
{
  Console.WriteLine(nameof(ShiftingBits));
  ushort s1 = 0b01;
  Console.WriteLine($"{"Binary",16} {"Decimal",8} {"Hex",6}");
  for (int i = 0; i < 16; i++)
  {
    Console.WriteLine($"{s1.ToBinaryString(),16} {s1,8} hex: {s1,6:X}");
    s1 = (ushort)(s1 << 1);
  }
  Console.WriteLine();
}
```

In the program output, you can see binary, decimal, and hexadecimal values with the loop:

```
          Binary  Decimal    Hex
0000000000000001        1      1
0000000000000010        2      2
0000000000000100        4      4
0000000000001000        8      8
0000000000010000       16     10
0000000000100000       32     20
0000000001000000       64     40
0000000010000000      128     80
0000000100000000      256    100
0000001000000000      512    200
0000010000000000     1024    400
0000100000000000     2048    800
0001000000000000     4096   1000
0010000000000000     8192   2000
0100000000000000    16384   4000
1000000000000000    32768   8000
```

Signed and Unsigned Numbers

One important thing to remember when working with binary numbers is that when using signed types, such as `int`, `long`, and `short`, the leftmost bit is used to represent the sign. When you use an `int`, the highest number available is `2147483647`—the positive number of 31 bits or `0x7FFF_FFFF`. With a `uint`, the highest number available is `4294967295` or `0xFFFF_FFFF`. This represents the positive number of 32 bits. With the `int`, the other half of the number range is used for negative numbers.

To understand how negative numbers are represented, the following code snippet initializes the `maxNumber` variable to the highest positive number that fits into 15 bits using `short.MaxValue`. Then, in a `for` loop, the variable is incremented three times. In the results, binary, decimal, and hexadecimal values are shown (code file `BinaryCalculations/Program.cs`):

```
void SignedNumbers()
{
  Console.WriteLine(nameof(SignedNumbers));

  void DisplayNumber(string title, short x) =>
    Console.WriteLine($"{title,-11} " +
      $"bin: {x.ToBinaryString().AddSeparators()}, " +
      $"dec: {x,6}, hex: {x,4:X}");
```

```
    short maxNumber = short.MaxValue;
    DisplayNumber("max short", maxNumber);
    for (int i = 0; i < 3; i++)
    {
      maxNumber++;
      DisplayNumber($"added {i + 1}", maxNumber);
    }
    Console.WriteLine();
    //...
}
```

With the output of the application, you can see all the bits—except the sign bit—are set to achieve the maximum integer value. The output shows the same value in different formats—binary, decimal, and hexadecimal. Adding 1 to the first output results in an overflow of the `short` type setting the sign bit, and all other bits are 0. This is the highest negative value for the `int` type. After this result, two more increments are done:

```
max short    bin: 0111_1111_1111_1111, dec:  32767, hex: 7FFF
added 1      bin: 1000_0000_0000_0000, dec: -32768, hex: 8000
added 2      bin: 1000_0000_0000_0001, dec: -32767, hex: 8001
added 3      bin: 1000_0000_0000_0010, dec: -32766, hex: 8002
```

With the next code snippet, the variable `zero` is initialized to `0`. In the `for` loop, this variable is decremented three times:

```
short zero = 0;
DisplayNumber("zero", zero);
for (int i = 0; i < 3; i++)
{
  zero--;
  DisplayNumber($"subtracted {i + 1}", zero);
}
Console.WriteLine();
```

With the output, you can see 0 is represented with all the bits not set. Doing a decrement results in decimal `-1`, which is all the bits set, including the sign bit:

```
zero         bin: 0000_0000_0000_0000, dec:      0, hex:    0
subtracted 1 bin: 1111_1111_1111_1111, dec:     -1, hex: FFFF
subtracted 2 bin: 1111_1111_1111_1110, dec:     -2, hex: FFFE
subtracted 3 bin: 1111_1111_1111_1101, dec:     -3, hex: FFFD
```

Next, start with the largest negative number for a `short`. The number is incremented three times:

```
short minNumber = short.MinValue;
DisplayNumber("min number", minNumber);
for (int i = 0; i < 3; i++)
{
  minNumber++;
  DisplayNumber($"added {i + 1}", minNumber);
}
Console.WriteLine();
```

The highest negative number was already shown earlier when overflowing the highest positive number. Earlier you saw this same number when `int.MinValue` was used. This number is then incremented three times:

```
min number   bin: 1000_0000_0000_0000, dec: -32768, hex: 8000
added 1      bin: 1000_0000_0000_0001, dec: -32767, hex: 8001
added 2      bin: 1000_0000_0000_0010, dec: -32766, hex: 8002
added 3      bin: 1000_0000_0000_0011, dec: -32765, hex: 8003
```

TYPE SAFETY

The Intermediate Language (IL) enforces strong type safety upon its code. Strong typing enables many of the services provided by .NET, including security and language interoperability. As you would expect from a language compiled into IL, C# is also strongly typed. Among other things, this means that data types are not always seamlessly interchangeable. This section looks at conversions between primitive types.

> **NOTE** *C# also supports conversions between different reference types and allows you to define how data types that you create behave when converted to and from other types. Both of these topics are discussed later in this chapter.*
>
> *Generics, however, enable you to avoid some of the most common situations in which you would need to perform type conversions. See Chapter 4, "Object-Oriented Programming in C#", and Chapter 8 when using many generic collection classes.*

Type Conversions

Often, you need to convert data from one type to another. Consider the following code:

```
byte value1 = 10;
byte value2 = 23;
byte total = value1 + value2;
Console.WriteLine(total);
```

When you attempt to compile these lines, you get the following error message:

```
Cannot implicitly convert type 'int' to 'byte'
```

The problem here is that when you add 2 bytes together, the result is returned as an `int`, not another `byte`. This is because a `byte` can contain only 8 bits of data, so adding 2 bytes together could easily result in a value that cannot be stored in a single `byte`. If you want to store this result in a `byte` variable, you have to convert it back to a `byte`. The following sections discuss two conversion mechanisms supported by C#—*implicit* and *explicit*.

Implicit Conversions

Conversion between types can normally be achieved automatically (implicitly) only if you can guarantee that the value is not changed in any way. This is why the previous code failed; by attempting a conversion from an `int` to a `byte`, you were potentially losing 3 bytes of data. The compiler won't let you do that unless you explicitly specify that's what you want to do. If you store the result in a `long` instead of a `byte`, however, you will have no problems:

```
byte value1 = 10;
byte value2 = 23;
long total = value1 + value2; // this will compile fine
Console.WriteLine(total);
```

Your program has compiled with no errors at this point because a `long` holds more bytes of data than a `byte`, so there is no risk of data being lost. In these circumstances, the compiler is happy to make the conversion for you without you needing to ask for it explicitly. As you would expect, you can perform implicit conversions only from a smaller integer type to a larger one, not from larger to smaller. You can also convert between integers and floating-point values; however, the rules are slightly different here. Though you can convert between types of the same size, such as `int`/`uint` to `float` and `long`/`ulong` to `double`, you can also convert from `long`/`ulong` to `float`. You might lose 4 bytes of data doing this, but it only means that the value of the `float` you receive will be

less precise than if you had used a `double`; the compiler regards this as an acceptable possible error because the magnitude of the value is not affected. You can also assign an unsigned variable to a signed variable as long as the value limits of the unsigned type fit between the limits of the signed variable.

Nullable value types introduce additional considerations when you're implicitly converting value types:

- Nullable value types implicitly convert to other nullable value types following the conversion rules described for non-nullable types in the previous rules; that is, `int?` implicitly converts to `long?`, `float?`, `double?`, and `decimal?`.
- Non-nullable value types implicitly convert to nullable value types according to the conversion rules described in the preceding rules; that is, `int` implicitly converts to `long?`, `float?`, `double?`, and `decimal?`.
- Nullable value types do not implicitly convert to non-nullable value types; you must perform an explicit conversion as described in the next section. That's because there is a chance that a nullable value type will have the value `null`, which cannot be represented by a non-nullable type.

Explicit Conversions

Many conversions cannot be implicitly made between types, and the compiler returns an error if any are attempted. The following are some of the conversions that cannot be made implicitly:

- `int` to `short`—Data loss is possible.
- `int` to `uint`—Data loss is possible.
- `uint` to `int`—Data loss is possible.
- `float` to `int`—Everything is lost after the decimal point.
- Any numeric type to `char`—Data loss is possible.
- `decimal` to any other numeric type—The decimal type is internally structured differently from both integers and floating-point numbers.
- `int?` to `int`—The nullable type may have the value `null`.

However, you can explicitly carry out such conversions using *casts*. When you cast one type to another, you deliberately force the compiler to make the conversion. A cast looks like this:

```
long val = 30000;
int i = (int)val; // A valid cast. The maximum int is 2147483647
```

You indicate the type to which you are casting by placing its name in parentheses before the value to be converted.

Casting can be a dangerous operation to undertake. Even a simple cast from a `long` to an `int` can cause problems if the value of the original `long` is greater than the maximum value of an `int`:

```
long val = 3000000000;
int i = (int)val; // An invalid cast. The maximum int is 2147483647
```

In this case, you get neither an error nor the result you expect. If you run this code and output the value stored in `i`, this is what you get:

```
-1294967296
```

It is good practice to assume that an explicit cast does not return the results you expect. As shown earlier, C# provides a `checked` operator that you can use to test whether an operation causes an arithmetic overflow. You can

use the `checked` operator to confirm that a cast is safe and to force the runtime to throw an overflow exception if it is not:

```
long val = 3000000000;
int i = checked((int)val);
```

Bearing in mind that all explicit casts are potentially unsafe, make sure you include code in your application to deal with possible failures of the casts. Chapter 10, "Errors and Exceptions," introduces structured exception handling using the `try` and `catch` statements.

Using casts, you can convert most primitive data types from one type to another; for example, in the following code, the value `0.5` is added to `price`, and the total is cast to an `int`:

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

This gives the price rounded to the nearest dollar. However, in this conversion, data is lost—namely, everything after the decimal point. Therefore, such a conversion should never be used if you want to continue to do more calculations using this modified price value. However, it is useful if you want to output the approximate value of a completed or partially completed calculation—if you don't want to bother the user with a lot of figures after the decimal point.

This example shows what happens if you convert an unsigned integer into a `char`:

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

The output is the character that has an ASCII number of 43, which is the + sign. This will work for any kind of conversion you want between the numeric types (including `char`), such as converting a `decimal` into a `char`, or vice versa.

Converting between value types is not restricted to isolated variables, as you have seen. You can convert an array element of type `double` to a struct member variable of type `int`:

```
struct ItemDetails
{
  public string Description;
  public int ApproxPrice;
}
//...
double[] Prices = { 25.30, 26.20, 27.40, 30.00 };
ItemDetails id;
id.Description = "Hello there.";
id.ApproxPrice = (int)(Prices[0] + 0.5);
```

To convert a nullable type to a non-nullable type or another nullable type where data loss may occur, you must use an explicit cast. This is true even when converting between elements with the same basic underlying type—for example, `int?` to `int` or `float?` to `float`. This is because the nullable type may have the value `null`, which cannot be represented by the non-nullable type. As long as an explicit cast between two equivalent non-nullable types is possible, so is the explicit cast between nullable types. However, when casting from a nullable type to a non-nullable type and the variable has the value `null`, an `InvalidOperationException` is thrown. Here is an example:

```
int? a = null;
int b = (int)a; // Will throw exception
```

By using explicit casts and a bit of care and attention, you can convert any instance of a simple value type to almost any other. However, there are limitations on what you can do with explicit type conversions—as far as value types are concerned, you can only convert to and from the numeric and `char` types and `enum` types. You cannot directly cast Booleans to any other type or vice versa.

If you need to convert between numeric and string, you can use methods provided in the .NET class library. The `Object` class implements a `ToString` method, which has been overridden in all the .NET predefined types and which returns a string representation of the object:

```
int i = 10;
string s = i.ToString();
```

Similarly, if you need to parse a string to retrieve a numeric or Boolean value, you can use the `Parse` method supported by all the predefined value types:

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); // Add 50 to prove it is really an int
```

Note that `Parse` registers an error by throwing an exception if it is unable to convert the string (for example, if you try to convert the string `Hello` to an integer). Again, exceptions are covered in Chapter 10. Instead of using the `Parse` method, you can also use `TryParse`, which doesn't throw an exception in case of an error, but returns `true` if it succeeds.

Boxing and Unboxing

Chapter 2 explains that all types—both the simple predefined types, such as `int` and `char`, and the complex types, such as classes and structs—derive from the `object` type. This means you can treat even literal values as though they are objects:

```
string s = 10.ToString();
```

However, you also saw that C# data types are divided into value types, which are allocated on the stack, and reference types, which are allocated on the managed heap. How does this work with the capability to call methods on an `int`, if the `int` is nothing more than a 4-byte value on the stack?

C# achieves this through a bit of magic called *boxing*. Boxing and its counterpart, *unboxing*, enable you to convert value types to reference types and then back to value types. I include this topic in the section on casting because this is essentially what you are doing—you are casting your value to the `object` type. Boxing is the term used to describe the transformation of a value type to a reference type. Basically, the runtime creates a temporary reference-type box for the object on the heap.

This conversion can occur implicitly, as in the preceding example, but you can also perform it explicitly:

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

Unboxing is the term used to describe the reverse process, whereby the value of a previously boxed value type is cast back to a value type. Here, I use the term *cast* because this has to be done explicitly. The syntax is similar to explicit type conversions already described:

```
int myIntNumber = 20;
object myObject = myIntNumber; // Box the int
int mySecondNumber = (int)myObject; // Unbox it back into an int
```

A variable can be unboxed only if it has been boxed. If you execute the last line when `myObject` is not a boxed `int`, you get a runtime exception.

One word of warning: When unboxing, you have to be careful that the receiving value is of the same type as the value that was boxed. Even if the resulting type has enough room to store all the bytes in the value being unboxed, an `InvalidCastException` is thrown. You can avoid this by casting from the original type in the new type, as shown here:

```
int myIntNumber = 42;
object myObject = (object)myIntNumber;
long myLongNumber = (long)(int)myObject;
```

OPERATOR OVERLOADING

Instead of invoking methods, the code can become more readable using operators. Just compare these two code lines to add two vectors:

```
vect3 = vect1 + vect2;
vect3 = vect1.Add(vect2);
```

With predefined number types, you can use `+`, `-`, `/`, `*`, and `%` operators, and you can also concatenate strings with the `+` operator. Using such operators is not only possible with predefined types, but also with custom types as long as they make sense with the types. What would a `+` operator used with two `Person` objects do?

You can overload the following operators:

OPERATORS	DESCRIPTION
`+x, -x, !x, ~x, ++, --, true, false`	These are unary operators that can be overloaded.
`x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y`	These are binary operators that can be overloaded.
`a[i], a?[i]`	Element access cannot be overloaded with an operator overload, but you can create an indexer, which is shown later in this chapter.
`(T)x`	Instead of using an operator overload, you can use the cast to create a user-defined conversion, which is shown later in this chapter as well.

> **NOTE** *You might wonder what the reason is for overloading the* `true` *and* `false` *operators. Conditional logical operators* `&&` *and* `||` *cannot be directly overloaded. To create a custom implementation for these operators, you can overload the* `true`*, the* `false`*, the* `&`*, and the* `|` *operators.*
>
> *Similarly, you can't explicitly overload compound conversion operators such as* `+=` *and* `-=`*. If you overload the binary operator, compound conversion is implicitly overloaded.*
>
> *Some operators need to be overloaded in pairs. If you overload* `==`*, you also must overload* `!=`*. If you overload* `<`*, then you must overload* `>`*, and if you overload* `<=`*, then you must overload* `>=`*.*

How Operators Work

To understand how to overload operators, it's useful to think about what happens when the compiler encounters an operator. Using the addition operator (`+`) as an example, suppose that the compiler processes the following lines of code:

```
int x = 1;
int y = 2;
long z = x + y;
```

The compiler identifies that it needs to add two integers and assign the result to a `long`. The expression `x + y` is just an intuitive and convenient syntax for calling a method that adds two numbers. The method takes two parameters, `x` and `y`, and returns their sum. Therefore, the compiler does the same thing it does for any method call: it looks for the best matching overload of the addition operator based on the parameter types—in this case, one that takes two integers. As with normal overloaded methods, the desired return type does not influence the compiler's choice as to which version of a method it calls. As it happens, the overload called in the example takes two `int` parameters and returns an `int`; this return value is subsequently converted to a `long`. This can result in an overflow if the two added `int` values don't fit into an `int` although a `long` is declared to write the result to.

The next lines cause the compiler to use a different overload of the addition operator:

```
double d1 = 4.0;
double d2 = d1 + x;
```

In this instance, the parameters are a `double` and an `int`, but there is no overload of the addition operator that takes this combination of parameters. Instead, the compiler identifies the best matching overload of the addition operator as being the version that takes two `double`s as its parameters, and it implicitly casts the `int` to a `double`. Adding two `double`s requires a different process from adding two integers. Floating-point numbers are stored as a mantissa and an exponent. Adding them involves bit-shifting the mantissa of one of the `double`s so that the two exponents have the same value, adding the mantissas, and then shifting the mantissa of the result and adjusting its exponent to maintain the highest possible accuracy in the answer.

Now you are in a position to see what happens if the compiler finds something like this:

```
Vector vect1, vect2, vect3;
// initialize vect1 and vect2
vect3 = vect1 + vect2;
vect1 = vect1 * 2;
```

Here, `Vector` is the struct, which is defined in the following section. The compiler sees that it needs to add two `Vector` instances, `vect1` and `vect2`, together. It looks for an overload of the addition operator, which takes two `Vector` instances as its parameters.

If the compiler finds an appropriate overload, it calls up the implementation of that operator. If it cannot find one, it checks whether there is any other overload for `+` that it can use as a best match—perhaps something with two parameters of other data types that can be implicitly converted to `Vector` instances. If the compiler cannot find a suitable overload, it raises a compilation error, just as it would if it could not find an appropriate overload for any other method call.

Operator Overloading with the Vector Type

This section demonstrates operator overloading through developing a struct named `Vector` that represents a three-dimensional vector. The 3D vector is just a set of three numbers (doubles) that tell you how far something is moving. The variables representing the numbers are called `X`, `Y`, and `Z`: the `X` tells you how far something moves east, `Y` tells you how far it moves north, and `Z` tells you how far it moves upward. Combine the three numbers and you get the total movement.

You can add or multiply vectors by other vectors or by numbers. Incidentally, in this context, we use the term *scalar*, which is math-speak for a simple number—in C# terms that is just a `double`. The significance of addition should be clear. If you move first by the vector `(3.0, 3.0, 1.0)` and then move by the vector `(2.0, -4.0, -4.0)`, the total amount you have moved can be determined by adding the two vectors. Adding vectors means adding each component individually, so you get `(5.0, -1.0, -3.0)`. In this context, mathematicians write `c=a+b`, where `a` and `b` are the vectors and `c` is the resulting vector. You want to be able to use the `Vector` struct the same way.

> **NOTE** *The fact that this example is developed as a struct rather than a class is not significant with operator overloading. Operator overloading works in the same way for structs, classes, and records.*

The following is the definition for `Vector`—containing the read-only public fields, constructors, and a `ToString` override so you can easily view the contents of a `Vector`. Operator overloads are added next (code file `OperatorOverloadingSample/Vector.cs`):

```
readonly struct Vector
{
  public Vector(double x, double y, double z) => (X, Y, Z) = (x, y, z);

  public Vector(Vector v) => (X, Y, Z) = (v.X, v.Y, v.Z);

  public readonly double X;
  public readonly double Y;
  public readonly double Z;
  public override string ToString() => $"( {X}, {Y}, {Z} )";
}
```

This example has two constructors that require specifying the initial value of the vector, either by passing in the values of each component or by supplying another `Vector` whose value can be copied. Constructors like the second one, which takes a single `Vector` argument, are often termed *copy constructors* because they effectively enable you to initialize a class or struct instance by copying another instance.

Here is the interesting part of the `Vector` struct—the operator overload that provides support for the addition operator:

```
public static Vector operator +(Vector left, Vector right) =>
  new Vector(left.X + right.X, left.Y + right.Y, left.Z + right.Z);
```

The operator overload is declared in much the same way as a static method, except that the `operator` keyword tells the compiler it is actually an operator overload you are defining. The `operator` keyword is followed by the actual symbol for the relevant operator, in this case the addition operator (`+`). The return type is whatever type you get when you use this operator. Adding two vectors results in a vector; therefore, the return type is also a `Vector`. For this particular override of the addition operator, the return type is the same as the containing class, but that is not necessarily the case, as you see later in this example. The two parameters are the things you are operating on. For binary operators (those that take two parameters), such as the addition and subtraction operators, the first parameter is the value on the left of the operator, and the second parameter is the value on the right.

The implementation of this operator returns a new `Vector` that is initialized using the `X`, `Y`, and `Z` fields from the `left` and `right` variables.

C# requires that all operator overloads be declared as `public` and `static`, which means they are associated with their class or struct, not with a particular instance. Because of this, the body of the operator overload has no access to nonstatic class members or the `this` identifier. This is fine because the parameters provide all the input data the operator needs to know to perform its task.

Now all you need to do is write some simple code to test the `Vector` struct (code file `OperatorOverloadingSample/Program.cs`):

```
Vector vect1, vect2, vect3;
vect1 = new(3.0, 3.0, 1.0);
vect2 = new(2.0, -4.0, -4.0);
vect3 = vect1 + vect2;
Console.WriteLine($"vect1 = {vect1}");
Console.WriteLine($"vect2 = {vect2}");
Console.WriteLine($"vect3 = {vect3}");
```

Compiling and running this code returns the following result:

```
vect1 = ( 3, 3, 1 )
vect2 = ( 2, -4, -4 )
vect3 = ( 5, -1, -3 )
```

Just by implementing the + operator, you can use the compound assignment operator +=. Let's add vect2 to the existing value of vect3:

```
vect3 += vect2;
Console.WriteLine($"vect3 = {vect3}");
```

This compiles and runs, resulting in the following:

```
vect3 = ( 7, -5, -7)
```

In addition to adding vectors, you can multiply and subtract them and compare their values. These operators can be implemented in the same way as the + operator. What might be more interesting is multiplying a vector by a double. With the following three operator overloads, a vector is multiplied by a vector, a vector is multiplied by a double, and a double is multiplied by a vector. You need to implement the different operators depending what's on the left and right sides, but you can reuse implementations. The operator overload where the vector is on the left and the double on the right just reuses the operator overload where the arguments are changed (code file OperatorOverloadingSample/Vector.cs):

```
public static Vector operator *(Vector left, Vector right) =>
  new Vector(left.X * right.X, left.Y * right.Y, left.Z * right.Z);

public static Vector operator *(double left, Vector right) =>
  new Vector(left * right.X, left * right.Y, left * right.Z);

public static Vector operator *(Vector left, double right) =>
  right * left;
```

The operators are used in the following code snippet. The int number used is converted to a double because this is the best match for the overload:

```
Console.WriteLine($"2 * vect3 = {2 * vect3}");
Console.WriteLine($"vect3 += vect2 gives {vect3 += vect2}");
Console.WriteLine($"vect3 = vect1 * 2 gives {vect3 = vect1 * 2}");
Console.WriteLine($"vect1 * vect3 = {vect1 * vect3}");
```

> **NOTE** *There's an important restriction on operator overloading. Because operator overloads are defined using a static member, you cannot add static members to an interface contract. This might change in a future C# version; interfaces just got an improvement with C# 8 and default interface methods. Some more improvements have already been discussed.*
>
> *In case you need operator overloads with generic types, you can create constraints on classes. The types can also be abstract classes and generic types. With generic types, you can implement operator overloads.*

COMPARING OBJECTS FOR EQUALITY

Comparing objects for equality has become easier with C# 9 and records. Records already have built-in functionality to compare the values of the type. Let's look at what's implemented with records (what you can override) and what you need to do with classes and structs.

To compare references, the object class defines the static method ReferenceEquals. This is not a comparison by value; instead it just compares the variables if they reference the same object in the heap. The functionality is the same for classes and records. Comparing two variables referencing the same object in the heap returns true. If the two variables reference different objects in the heap, the method returns false, even if the content of the two objects is the same. Using this method to compare two variables referencing structs, new objects are created

to reference the value type (known as *boxing*) and thus always returns `false`. The compiler warns on comparing structs this way.

The default implementation of the `object` class `Equals` method just invokes `object.ReferenceEquals`. In case you need to compare the values for equality, you can use the built-in functionality of the record type or create a custom implementation with the class. To compare the values of two reference types, you need to consider what's automatically implemented by a record and what you can implement when comparing classes for equality:

- The object type defines the virtual method `bool Equals(object?)` that can be overridden.
- The interface `IEquatable<T>` defines the generic method `bool Equals(T? object)` that can be implemented.
- The operators `==` and `!=` can be overridden.
- Records also implement an `EqualityContract`, which is used with the comparison to not only compare the values, but also if the comparison is done with the same contract.

To compare references, the `Book` class implements the `IEquatable<Book>` interface with the `bool Equals(Book? other)` method. This method compares the `Title` and `Publisher` properties. Similar to the record type, the `Book` class specifies the `EqualityContract` property to also compare the type of the class. This way, comparing the `Title` and `Publisher` properties with an object of another type returns always `false`. The implementation for equality comparison is only done with this method. The overridden `Equals` method from the base class invokes this method, as well as the implementation for the operators `==` and `!=`. Implementing equality also requires overriding the `GetHashCode` method from the base class (code file `EqualitySample/Book.cs`):

```
class Book : IEquatable<Book>
{
  public Book(string title, string publisher)
  {
    Title = title;
    Publisher = publisher;
  }
  public string Title { get; }
  public string Publisher { get; }

  protected virtual Type EqualityContract { get; } = typeof(Book);

  public override string ToString() => Title;

  public override bool Equals(object? obj) =>
    this == obj as Book;

  public override int GetHashCode() =>
    Title.GetHashCode() ^ Publisher.GetHashCode();

  public virtual bool Equals(Book? other) =>
    this == other;

  public static bool operator ==(Book? left, Book? right) =>
    left?.Title == right?.Title && left?.Publisher == right?.Publisher &&
    left?.EqualityContract == right?.EqualityContract;

  public static bool operator !=(Book? left, Book? right) =>
    !(left == right);
}
```

> **NOTE** *Don't be tempted to overload the comparison operator by only calling the instance version of the* `Equals` *method inherited from* `System.Object`. *If you do so and then an attempt is made to evaluate* `(objA == objB)`, *when* `objA` *happens to be* `null`, *you get an exception because the .NET runtime tries to evaluate* `null.Equals(objB)`. *Working the other way around (overriding* `Equals` *to call the comparison operator) should be safe.*

> **NOTE** *To implement equality comparison, some work is needed. With the record type, this work is done from the compiler. If you use records within records, everything works out of the box. However, if you use classes as record members, only references are compared—unless you implement equality comparison.*

In the `Program.cs` file, two `Book` objects are created that have the same content. Because there are two different objects in the heap, `object.ReferenceEquals` returns `false`. Next, the `Equals` method from the `IEquatable<Book>` interface, the overloaded `object Equals`, and the operator `==` are used, and they all return `true` because of the implemented value comparison (code file `EqualitySample/Program.cs`):

```
Book book1 = new("Professional C#", "Wrox Press");
Book book2 = new("Professional C#", "Wrox Press");

if (!object.ReferenceEquals(book1, book2))
{
  Console.WriteLine("Not the same reference");
}

if (book1.Equals(book2))
{
  Console.WriteLine("The same object using the generic Equals method");
}

object book3 = book2;
if (book1.Equals(book3))
{
  Console.WriteLine("The same object using the overridden Equals method");
}

if (book1 == book2)
{
  Console.WriteLine("The same book using the == operator");
}
```

> **NOTE** *For struct types, similar functionality to classes applies; there are just a few important differences. Remember, you can't use* `object.ReferenceEquals` *with value types. Another difference is that the* `object.Equals` *method is already overridden to compare the values. For more functionality with equality, similar to what has been shown with the* `Book` *class, implement the* `IEquality<T>` *interface and override the* `==` *and* `!=` *operators.*

IMPLEMENTING CUSTOM INDEXERS

Custom indexers cannot be implemented using the operator overloading syntax, but they can be implemented with a syntax that looks similar to properties.

With the following code snippet, an array is created, and the indexer is used to access array elements. The second code line uses the indexer to access the second element and pass `42` to it. The third line uses the indexer to access the third element and pass the value of the element to the variable `x`.

```
int[] arr1 = {1, 2, 3};
arr1[1] = 42;
int x = arr1[2];
```

> **NOTE** *Arrays are explained in Chapter 6.*

To create a custom indexer, first create a `Person` record with the properties `FirstName`, `LastName`, and `Birthday` (code file `CustomIndexerSample/Person.cs`):

```
public record Person(string FirstName, string LastName, DateTime Birthday)
{
  public override string ToString() => $"{FirstName} {LastName}";
}
```

The class `PersonCollection` defines a private array field that contains `Person` elements and a constructor where a number of `Person` objects can be passed (code file `CustomIndexerSample/PersonCollection.cs`):

```
public class PersonCollection
{
  private Person[] _people;

  public PersonCollection(params Person[] people) =>
    _people = people.ToArray();
}
```

For allowing indexer-syntax to be used to access the `PersonCollection` and return `Person` objects, you can create an indexer. The indexer looks very similar to a property because it also contains `get` and `set` accessors. What's different is the name. Specifying an indexer makes use of the `this` keyword. The brackets that follow the `this` keyword specify the type that is used with the index. An array offers indexers with the `int` type, so `int` types are used here to pass the information directly to the contained array `_people`. The use of the `set` and `get` accessors is similar to properties. The `get` accessor is invoked when a value is retrieved; the `set` accessor is invoked when a `Person` object is passed on the right side.

```
public Person this[int index]
{
  get => _people[index];
  set => _people[index] = value;
}
```

With indexers, any type can be used as the indexing type. With the sample application, the `DateTime` struct is used. This indexer is used to return every person with a specified birthday. Because multiple people can have the same birthday, not a single `Person` object is returned; instead, a list of people is returned with the interface `IEnumerable<Person>`. With the implementation of the indexer, the `Where` method is used. A lambda expression is passed with the argument. The `Where` method is defined in the namespace `System.Linq`:

```
public IEnumerable<Person> this[DateTime birthDay]
{
  get => _people.Where(p => p.Birthday == birthDay);
}
```

The indexer using the `DateTime` type lets you retrieve `Person` objects but doesn't allow you to set `Person` objects because there's a `get` accessor but no `set` accessor. A shorthand notation exists to create the same code with an expression-bodied member (the same syntax available with properties):

```
public IEnumerable<Person> this[DateTime birthDay] =>
  _people.Where(p => p.Birthday == birthDay);
```

With the top-level statements of the sample application, a `PersonCollection` object with four `Person` objects is created. With the first `WriteLine` method, the third element is accessed using the `get` accessor of the indexer with the `int` parameter. Within the `foreach` loop, the indexer with the `DateTime` parameter is used to pass a specified date (code file `CustomIndexerSample/Program.cs`):

```
Person p1 = new("Ayrton", "Senna", new DateTime(1960, 3, 21));
Person p2 = new("Ronnie", "Peterson", new DateTime(1944, 2, 14));
Person p3 = new("Jochen", "Rindt", new DateTime(1942, 4, 18));
Person p4 = new("Francois", "Cevert", new DateTime(1944, 2, 25));
PersonCollection coll = new(p1, p2, p3, p4);
Console.WriteLine(coll[2]);
foreach (var r in coll[new DateTime(1960, 3, 21)])
{
  Console.WriteLine(r);
}
Console.ReadLine();
```

When you run the program, the first `WriteLine` method writes `Jochen Rindt` to the console; the result of the `foreach` loop is `Ayrton Senna` because that person has the same birthday as is assigned within the second indexer.

USER-DEFINED CONVERSIONS

Earlier in this chapter (see the "Explicit Conversions" section), you learned that you can convert values between predefined data types through a process of casting. You also saw that C# allows two different types of casts: implicit and explicit. This section looks at these types of casts.

For an explicit cast, you explicitly mark the cast in your code by including the destination data type inside parentheses:

```
int i = 3;
long l = i; // implicit
short s = (short)i; // explicit
```

For the predefined data types, explicit casts are required where there is a risk that the cast might fail or some data might be lost. The following are some examples:

- When converting from an `int` to a `short`, the `short` might not be large enough to hold the value of the `int`.
- When converting from signed to unsigned data types, incorrect results are returned if the signed variable holds a negative value.
- When converting from floating-point to integer data types, the fractional part of the number will be lost.
- When converting from a nullable type to a non-nullable type, a value of `null` causes an exception.

By making the cast explicit in your code, C# forces you to affirm that you understand there is a risk of data loss, and therefore presumably you have written your code to take this into account.

Because C# allows you to define your own data types (structs and classes), it follows that you need the facility to support casts to and from those data types. The mechanism is to define a cast as a member operator of one of the relevant classes. Your cast operator must be marked as either `implicit` or `explicit` to indicate how you intend it to be used. The expectation is that you follow the same guidelines as for the predefined casts: if you know that the cast is always safe regardless of the value held by the source variable, then you define it as `implicit`. Conversely, if you know there is a risk of something going wrong for certain values—perhaps some loss of data or an exception being thrown—then you should define the cast as `explicit`.

> **NOTE** *You should define any custom casts you write as explicit if there are any source data values for which the cast will fail or if there is any risk of an exception being thrown.*

The syntax for defining a cast is similar to that for overloading operators discussed earlier in this chapter. This is not a coincidence—a cast is regarded as an operator whose effect is to convert from the source type to the destination type. To illustrate the syntax, the following is taken from an example `struct` named `Currency`, which is introduced in the next section, "Implementing User-Defined Casts":

```
public static implicit operator float (Currency value)
{
  // processing
}
```

The return type of the operator defines the target type of the cast operation, and the single parameter is the source object for the conversion. The cast defined here allows you to implicitly convert the value of a `Currency` into a `float`. Note that if a conversion has been declared as `implicit`, the compiler permits its use either implicitly or explicitly. If it has been declared as `explicit`, the compiler only permits it to be used explicitly. Similar to other operator overloads, casts must be declared as both `public` and `static`.

Implementing User-Defined Casts

This section illustrates the use of implicit and explicit user-defined casts in an example called `CastingSample`. In this example, you define a struct, `Currency`, which holds a positive USD ($) monetary value. C# provides the `decimal` type for this purpose, but it is possible you still will want to write your own struct or class to represent monetary values if you need to perform sophisticated financial processing and therefore want to implement specific methods on such a class.

> **NOTE** *The syntax for casting is the same for structs and classes. This example happens to be for a struct, but it would work just as well if you declared* `Currency` *as a class.*

Initially, the definition of the `Currency` struct is as follows (code file `CastingSample/Currency.cs`):

```
public readonly struct Currency
{
  public readonly uint Dollars;
  public readonly ushort Cents;
```

```
  public Currency(uint dollars, ushort cents) => (Dollars, Cents) = (dollars, cents);

  public override string ToString() => $"${Dollars}.{Cents,-2:00}";
}
```

The use of unsigned data types for the `Dollar` and `Cents` fields ensures that a `Currency` instance can hold only positive values. It is restricted this way to illustrate some points about explicit casts later. You might want to use a type like this to hold, for example, salary information for company employees (people's salaries tend not to be negative!).

Start by assuming that you want to be able to convert `Currency` instances to `float` values, where the integer part of the `float` represents the dollars. In other words, you want to be able to write code like this:

```
Currency balance = new(10, 50);
float f = balance; // We want f to be set to 10.5
```

To be able to do this, you need to define a cast. Hence, you add the following to your `Currency` definition:

```
public static implicit operator float (Currency value) =>
  value.Dollars + (value.Cents/100.0f);
```

The preceding cast is implicit. It is a sensible choice in this case because, as it should be clear from the definition of `Currency`, any value that can be stored in the `Currency` can also be stored in a `float`. There is no way that anything should ever go wrong in this cast.

> **NOTE** *There is a slight cheat here. In fact, when converting a* `uint` *to a* `float`*, there can be a loss in precision, but Microsoft has deemed this error sufficiently marginal to count the* `uint`*-to-*`float` *cast as implicit.*

However, if you have a `float` that you would like to be converted to a `Currency`, the conversion is not guaranteed to work. A `float` can store negative values, whereas `Currency` instances can't, and a `float` can store numbers of a far higher magnitude than can be stored in the `(uint) Dollar` field of `Currency`. Therefore, if a `float` contains an inappropriate value, converting it to a `Currency` could give unpredictable results. Because of this risk, the conversion from `float` to `Currency` should be defined as explicit. Here is the first attempt, which does not return quite the correct results, but it is instructive to examine why:

```
public static explicit operator Currency (float value)
{
  uint dollars = (uint)value;
  ushort cents = (ushort)((value-dollars)*100);
  return new Currency(dollars, cents);
}
```

The following code now successfully compiles:

```
float amount = 45.63f;
Currency amount2 = (Currency)amount;
```

However, the following code, if you tried it, would generate a compilation error because it attempts to use an explicit cast implicitly:

```
float amount = 45.63f;
Currency amount2 = amount; // wrong
```

By making the cast explicit, you warn the developer to be careful because data loss might occur. However, as you will soon see, this is not how you want your `Currency` struct to behave. Try writing a test harness and running

the sample. Here is the `Main` method, which instantiates a `Currency` struct and attempts a few conversions. At the start of this code, you write out the value of `balance` in two different ways—this is needed to illustrate something later in the example (code file `CastingSample/Program.cs`):

```
try
{
  Currency balance = new(50,35);
  Console.WriteLine(balance);
  Console.WriteLine($"balance is {balance}"); // implicitly invokes ToString
  float balance2 = balance;
  Console.WriteLine($"After converting to float, = {balance2}");
  balance = (Currency) balance2;
  Console.WriteLine($"After converting back to Currency, = {balance}");
  Console.WriteLine("Now attempt to convert out of range value of " +
    "-$50.50 to a Currency:");

  checked
  {
    balance = (Currency) (-50.50);
    Console.WriteLine($"Result is {balance}");
  }
}
catch(Exception e)
{
  Console.WriteLine($"Exception occurred: {e.Message}");
}
```

Notice that the entire code is placed in a `try` block to catch any exceptions that occur during your casts. In addition, the lines that test converting an out-of-range value to `Currency` are placed in a `checked` block in an attempt to trap negative values. Running this code produces the following output:

```
50.35
Balance is $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$50.50 to a Currency:
Result is $4294967246.00
```

This output shows that the code did not quite work as expected. First, converting back from `float` to `Currency` gave a wrong result of `$50.34` instead of `$50.35`. Second, no exception was generated when you tried to convert an obviously out-of-range value.

The first problem is caused by rounding errors. If a cast is used to convert from a `float` to a `uint`, the computer truncates the number rather than rounds it. The computer stores numbers in binary rather than decimal, and the fraction 0.35 cannot be exactly represented as a binary fraction (just as 1⁄3 cannot be represented exactly as a decimal fraction; it comes out as 0.3333 recurring). The computer ends up storing a value very slightly lower than 0.35 that can be represented exactly in binary format. Multiply by 100, and you get a number fractionally less than 35, which is truncated to 34 cents. Clearly, in this situation, such errors caused by truncation are serious, and the way to avoid them is to ensure that some intelligent rounding is performed in numerical conversions.

Luckily, Microsoft has written a class that does this: `System.Convert`. The `System.Convert` object contains a large number of static methods to perform various numerical conversions, and the one that we want is `Convert.ToUInt16`. Note that the extra care taken by the `System.Convert` methods comes at a performance cost. You should use them only when necessary.

Let's examine the second problem—why the expected overflow exception wasn't thrown. The issue here is that the place where the overflow really occurs isn't actually in the `Main` routine at all—it is inside the

code for the cast operator, which is called from the `Main` method. The code in this method was not marked as `checked`.

The solution is to ensure that the cast itself is computed in a `checked` context, too. With both this change and the fix for the first problem, the revised code for the conversion looks like the following:

```
public static explicit operator Currency (float value)
{
  checked
  {
    uint dollars = (uint)value;
    ushort cents = Convert.ToUInt16((value-dollars)*100);
    return new Currency(dollars, cents);
  }
}
```

Note that you use `Convert.ToUInt16` to calculate the cents, as described earlier, but you do not use it for calculating the dollar part of the amount. `System.Convert` is not needed when calculating the dollar amount because truncating the `float` value is what you want there.

> **NOTE** *The* `System.Convert` *methods also carry out their own overflow checking. Hence, for the particular case we are considering, there is no need to place the call to* `Convert.ToUInt16` *inside the checked context. The checked context is still required, however, for the explicit casting of* `value` *to dollars.*

You won't look at the new results with this new `checked` cast just yet because you have some more modifications to make to the `CastingSample` example later in this section.

> **NOTE** *If you are defining a cast that will be used very often, and for which performance is at an absolute premium, you may prefer not to do any error checking. That is also a legitimate solution, provided that the behavior of your cast and the lack of error checking are very clearly documented.*

Casts Between Classes

The `Currency` example involves only classes that convert to or from `float`—one of the predefined data types. However, it is not necessary to involve any of the simple data types. It is perfectly legitimate to define casts to convert between instances of different structs or classes that you have defined. You need to be aware of a couple of restrictions, however:

- You cannot define a cast if one of the classes is derived from the other (these types of casts already exist, as you will see later).
- The cast must be defined inside the definition of either the source or the destination data type.

To illustrate these requirements, suppose that you have the class hierarchy shown in Figure 5-1.

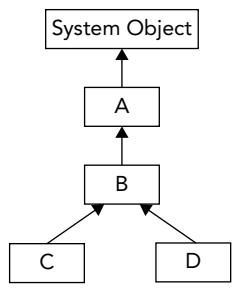

FIGURE 5-1

In other words, classes `C` and `D` are indirectly derived from `A`. In this case, the only legitimate user-defined cast between `A`, `B`, `C`, or `D` would be to convert between classes `C` and `D`, because these classes are not derived from each other. The code for this might look like the following (assuming you want the casts to be explicit, which is usually the case when defining casts between user-defined classes):

```
public static explicit operator D(C value)
{
  //...
}

public static explicit operator C(D value)
{
  //...
}
```

For each of these casts, you can choose where you place the definitions—inside the class definition of `C` or inside the class definition of `D`, but not anywhere else. C# requires you to put the definition of a cast inside either the source class (or struct) or the destination class (or struct). A side effect of this is that you cannot define a cast between two classes unless you have access to edit the source code for at least one of them. This is sensible because it prevents third parties from introducing casts into your classes.

After you have defined a cast inside one of the classes, you cannot also define the same cast inside the other class. Obviously, there should be only one cast for each conversion; otherwise, the compiler would not know which one to use.

Casts Between Base and Derived Classes

To see how these casts work, start by considering the case in which both the source and the destination are reference types and consider two classes, `MyBase` and `MyDerived`, where `MyDerived` is derived directly or indirectly from `MyBase`.

First, from `MyDerived` to `MyBase`, it is always possible (assuming the constructors are available) to write this:

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

Here, you are casting implicitly from `MyDerived` to `MyBase`. This works because of the rule that any reference to a type `MyBase` is allowed to refer to objects of class `MyBase` or anything derived from `MyBase`. In object-oriented programming, instances of a derived class are, in a real sense, instances of the base class, plus something extra. All the functions and fields defined on the base class are defined in the derived class, too.

Alternatively, you can write this:

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject; // OK
MyDerived derivedCopy2 = (MyDerived) baseObject; // Throws exception
```

This code is perfectly legal C# (in a syntactic sense, that is) and illustrates casting from a base class to a derived class. However, the final statement throws an exception when executed. When you perform the cast, the object being referred to is examined. Because a base class reference can, in principle, refer to a derived class instance, it is possible that this object is actually an instance of the derived class that you are attempting to cast to. If that is the case, the cast succeeds, and the derived reference is set to refer to the object. If, however, the object in question is not an instance of the derived class (or of any class derived from it), the cast fails, and an exception is thrown.

Notice that the casts that the compiler has supplied, which convert between base and derived class, do not actually do any data conversion on the object in question. All they do is set the new reference to refer to the object if it is legal for that conversion to occur. To that extent, these casts are very different in nature from the ones that you normally define yourself. For example, in the `CastingSample` example earlier, you defined casts that convert between a `Currency` struct and a `float`. In the `float`-to-`Currency` cast, you actually instantiated a new

Currency struct and initialized it with the required values. The predefined casts between base and derived classes do not do this. If you want to convert a MyBase instance into a real MyDerived object with values based on the contents of the MyBase instance, you cannot use the cast syntax to do this. The most sensible option is usually to define a derived class constructor that takes a base class instance as a parameter and have this constructor perform the relevant initializations:

```
class DerivedClass: BaseClass
{
  public DerivedClass(BaseClass base)
  {
    // initialize object from the Base instance
  }
  // ...
```

Boxing and Unboxing Casts

The previous discussion focused on casting between base and derived classes where both participants were reference types. Similar principles apply when casting value types, although in this case it is not possible to simply copy references—some copying of data must occur.

It is not, of course, possible to derive from structs or primitive value types. Casting between base and derived structs invariably means casting between a primitive type or a struct and System.Object. (Theoretically, it is possible to cast between a struct and System.ValueType, though it is hard to see why you would want to do this.)

The cast from any struct (or primitive type) to object is always available as an implicit cast—because it is a cast from a derived type to a base type—and is just the familiar process of boxing. Here's an example using the Currency struct:

```
Currency balance = new(40,0);
object baseCopy = balance;
```

When this implicit cast is executed, the contents of balance are copied onto the heap into a boxed object, and the baseCopy object reference is set to this object. What actually happens behind the scenes is this: when you originally defined the Currency struct, .NET implicitly supplied another (hidden) class, a boxed Currency class, which contains all the same fields as the Currency struct but is a reference type, stored on the heap. This happens whenever you define a value type, whether it is a struct or an enum, and similar boxed reference types exist corresponding to all the primitive value types of int, double, uint, and so on. It is not possible, or necessary, to gain direct programmatic access to any of these boxed classes in source code, but they are the objects that are working behind the scenes whenever a value type is cast to object. When you implicitly cast Currency to object, a boxed Currency instance is instantiated and initialized with all the data from the Currency struct. In the preceding code, it is this boxed Currency instance to which baseCopy refers. By these means, it is possible for casting from derived to base type to work syntactically in the same way for value types as for reference types.

Casting the other way is known as *unboxing*. Like casting between a base reference type and a derived reference type, it is an explicit cast because an exception is thrown if the object being cast is not of the correct type:

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject; // OK
Currency derivedCopy2 = (Currency)baseObject; // Exception thrown
```

This code works in a way similar to the code presented earlier for reference types. Casting derivedObject to Currency works fine because derivedObject actually refers to a boxed Currency instance—the cast is performed by copying the fields out of the boxed Currency object into a new Currency struct. The second cast fails because baseObject does not refer to a boxed Currency object.

When using boxing and unboxing, it is important to understand that both processes actually copy the data into the new boxed or unboxed object. Hence, manipulations on the boxed object, for example, do not affect the contents of the original value type.

Multiple Casting

One thing you have to watch for when you are defining casts is that if the C# compiler is presented with a situation in which no direct cast is available to perform a requested conversion, it attempts to find a way of combining casts to do the conversion. For example, with the `Currency` struct, suppose the compiler encounters a few lines of code like this:

```
Currency balance = new(10,50);
long amount = (long)balance;
double amountD = balance;
```

You first initialize a `Currency` instance, and then you attempt to convert it to a `long`. The trouble is that you haven't defined the cast to do that. However, this code still compiles successfully. Here's what happens: the compiler realizes that you have defined an implicit cast to get from `Currency` to `float`, and the compiler already knows how to explicitly cast a `float` to a `long`. Hence, it compiles that line of code into IL code that converts `balance` first to a `float` and then converts that result to a `long`. The same thing happens in the final line of the code, when you convert `balance` to a `double`. However, because the cast from `Currency` to `float` and the predefined cast from `float` to `double` are both implicit, you can write this conversion in your code as an implicit cast. If you prefer, you could also specify the casting route explicitly:

```
Currency balance = new(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

However, in most cases, this would be seen as needlessly complicating your code. The following code, by contrast, produces a compilation error:

```
Currency balance = new(10,50);
long amount = balance;
```

The reason is that the best match for the conversion that the compiler can find is still to convert first to `float` and then to `long`. The conversion from `float` to `long` needs to be specified explicitly, though.

Not all of this by itself should give you too much trouble. The rules are, after all, fairly intuitive and designed to prevent any data loss from occurring without the developer knowing about it. However, the problem is that if you are not careful when you define your casts, it is possible for the compiler to select a path that leads to unexpected results. For example, suppose that it occurs to someone else in the group writing the `Currency` struct that it would be useful to be able to convert a `uint` containing the total number of cents in an amount into a `Currency` (cents, not dollars, because the idea is not to lose the fractions of a dollar). Therefore, this cast might be written to try to achieve this:

```
// Do not do this!
public static implicit operator Currency(uint value) =>
  new Currency(value/100u, (ushort)(value%100));
```

Note the `u` after the first 100 in this code ensures that `value/100u` is interpreted as a `uint`. If you had written `value/100`, the compiler would have interpreted this as an `int`, not a `uint`.

The comment `Do not do this!` is clearly noted in this code, and here is why: the following code snippet merely converts a `uint` containing `350` into a `Currency` and back again; but what do you think `bal2` will contain after executing this?

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

The answer is not `350` but `3`! Moreover, it all follows logically. You convert `350` implicitly to a `Currency`, giving the result `balance.Dollars = 3`, `balance.Cents = 50`. Then the compiler does its usual figuring out of the best path for the conversion back. `Balance` ends up being implicitly converted to a `float` (value `3.5`), and this is converted explicitly to a `uint` with value `3`. One way to fix this would be to create a user-defined cast to `uint`.

Of course, other instances exist in which converting to another data type and back again causes data loss. For example, converting a `float` containing `5.8` to an `int` and back to a `float` again loses the fractional part, giving you a result of `5`, but there is a slight difference in principle between losing the fractional part of a number and dividing an integer by more than 100. `Currency` has suddenly become a rather dangerous class that does strange things to integers!

The problem is that there is a conflict between how your casts interpret integers. The casts between `Currency` and `float` interpret an integer value of `1` as corresponding to one dollar, but the latest `uint`-to-`Currency` cast interprets this value as one cent. This is an example of poor design. If you want your classes to be easy to use, you should ensure that all your casts behave in ways that are mutually compatible, in the sense that they intuitively give the same results. In this case, the solution is obviously to rewrite the `uint`-to-`Currency` cast so that it interprets an integer value of `1` as one dollar:

```
public static implicit operator Currency (uint value) =>
  new Currency(value, 0);
```

Incidentally, you might wonder whether this new cast is necessary at all. The answer is that it could be useful. Without this cast, the only way for the compiler to carry out a `uint`-to-`Currency` conversion would be via a `float`. Converting directly is a lot more efficient in this case, so having this extra cast provides performance benefits, though you need to ensure that it provides the same result as via a `float`, which you have now done. In other situations, you may also find that separately defining casts for different predefined data types enables more conversions to be implicit rather than explicit, though that is not the case here.

A good test of whether your casts are compatible is to ask whether a conversion will give the same results (other than perhaps a loss of accuracy as in `float`-to-`int` conversions) regardless of which path it takes. The `Currency` class provides a good example of this. Consider this code:

```
Currency balance = new(50, 35);
ulong bal = (ulong) balance;
```

At present, there is only one way that the compiler can achieve this conversion: by converting the `Currency` to a `float` implicitly and then to a `ulong` explicitly. The `float`-to-`ulong` conversion requires an explicit conversion, but that is fine because you have specified one here.

Suppose, however, that you then added another cast to convert implicitly from a `Currency` to a `uint`. You actually do this by modifying the `Currency` struct by adding the casts both to and from `uint` (code file `CastingSample/Currency.cs`):

```
public static implicit operator Currency(uint value) =>
  new Currency(value, 0);
public static implicit operator uint(Currency value) => value.Dollars;
```

Now the compiler has another possible route to convert from `Currency` to `ulong`: to convert from `Currency` to `uint` implicitly and then to `ulong` implicitly. Which of these two routes will it take? C# has some precise rules about the best route for the compiler when there are several possibilities. (The rules are not covered in this book, but if you are interested in the details, see the MSDN documentation.) The best answer is that you should design your casts so that all routes give the same answer (other than possible loss of precision), in which case it doesn't really matter which one the compiler picks. (As it happens in this case, the compiler picks the `Currency`-to-`uint`-to-`ulong` route in preference to `Currency`-to-`float`-to-`ulong`.)

To test casting the `Currency` to `uint`, add this test code to the `Main` method (code file `UserDefinedConversion/Program.cs`):

```
try
{
  Currency balance = new(50,35);
  Console.WriteLine(balance);
  Console.WriteLine($"balance is {balance}");
  uint balance3 = (uint) balance;
  Console.WriteLine($"Converting to uint gives {balance3}");
}
```

```
catch (Exception ex)
{
  Console.WriteLine($"Exception occurred: {ex.Message}");
}
```

Running the sample now gives you these results:

```
50
balance is $50.35
Converting to uint gives 50
```

The output shows that the conversion to `uint` has been successful, though, as expected, you have lost the cents part of the `Currency` in making this conversion.

However, the output also demonstrates one last potential problem that you need to be aware of when working with casts. The first line of output does not display the balance correctly, displaying `50` instead of `50.35`.

So, what is going on? The problem here is that when you combine casts with method overloads, you get another source of unpredictability.

The `WriteLine` statement using the format string implicitly calls the `Currency.ToString` method, ensuring that the `Currency` is displayed as a string.

The first code line with `WriteLine`, however, simply passes a raw `Currency` struct to the `WriteLine` method. Now, `WriteLine` has many overloads, but none of them takes a `Currency` struct. Therefore, the compiler starts fishing around to see what it can cast the `Currency` to in order to make it match up with one of the overloads of `WriteLine`. As it happens, one of the `WriteLine` overloads is designed to display `uint`s quickly and efficiently, and it takes a `uint` as a parameter—you have now supplied a cast that converts `Currency` implicitly to `uint`.

In fact, `WriteLine` has another overload that takes a `float` as a parameter and displays the value of that `float`. If you look closely at the output running the example previously where the cast to `uint` did not exist, you see that the first line of output displayed `Currency` as a `float`, using this overload. In that example, there wasn't a direct cast from `Currency` to `uint`, so the compiler picked `Currency`-to-`float` as its preferred way of matching up the available casts to the available `WriteLine` overloads. However, now that there is a direct cast to `uint` available in `Currency`, the compiler has opted for that route.

The upshot of this is that if you have a method call that takes several overloads and you attempt to pass it a parameter whose data type doesn't match any of the overloads exactly, then you are forcing the compiler to decide not only what casts to use to perform the data conversion, but also which overload, and hence which data conversion, to pick. The compiler always works logically and according to strict rules, but the results may not be what you expect. If there is any doubt, you are better off specifying which cast to use explicitly.

SUMMARY

This chapter looked at the standard operators provided by C#, described the mechanics of object equality, and examined how the compiler converts the standard data types from one to another. It also demonstrated how you can implement custom operator support on your data types using operator overloads. Finally, you looked at a special type of operator overload, the cast operator, which enables you to specify how instances of your types are converted to other data types.

The next chapter dives into arrays where the index operator has an important role.